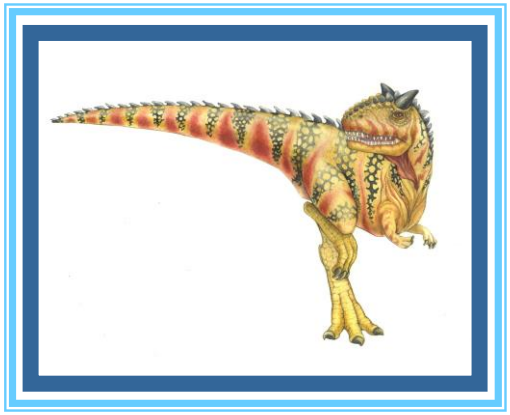
Chapter 6: Process Synchronization



Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-. Moderate-, and high-contention scenarios

Background

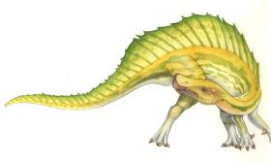
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

What's the problem?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers.
- We can do so by having a shared **`counter`** that keeps track of the number of full buffers.
- Initially, **`counter`** is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - It is decremented by the consumer after it consumes a buffer.
- If this is written in C++, what is the code to increment/decrement the counter?

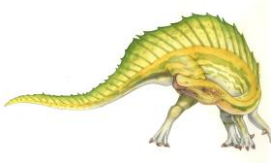
Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        counter++;
        in = (in + 1) % BUFFER_SIZE;
}
```



Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```



Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
mov  ecx, [counter]
inc  ecx
mov  [counter], ecx
```

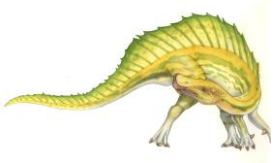
- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

```
mov  ecx, [counter]
dec  ecx
mov  [counter], ecx
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

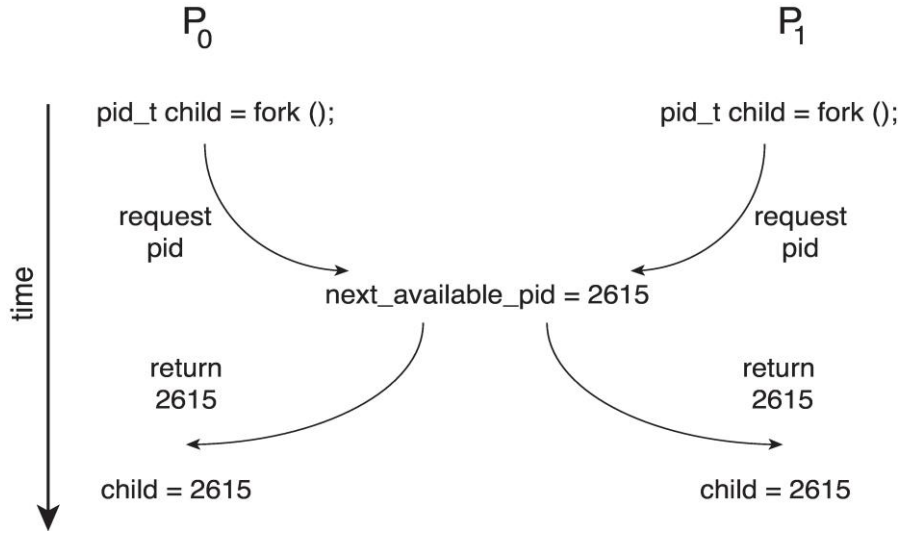


Critical Section

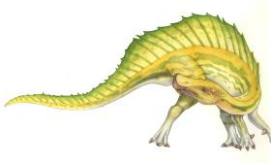
- Any code segment that manipulates a shared variable should be part of a **critical section**.
- In the producer-consumer example, `counter` should be incremented/decremented in a critical section.
- A critical section is a section that must be synchronized for multiple processes or threads because lack of synchronization can result in errors.
- The simplest case is shared variables, but other shared resources need to be synchronized as well. Often the code to allocate / return the resource is part of a critical section.

Race Condition

- Processes P_0 and P_1 are creating child process using `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!
- Access to `next_available_pid` must be an **atomic** operation.



Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

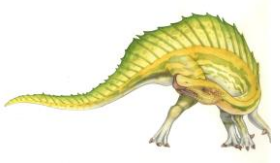
Critical Section

- General structure of process P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes



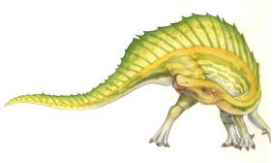
Solution Synonyms

- **Mutual Exclusion**
  - Mutual Exclusion guarantees that results are correct and not corrupted. Each process has access to the critical region and is the only process that has access at that time
- **Deadlock** happens when one or more processes can't make progress.
  - Deadlock generally involves two or more processes vying for the same resources where P_i has R_i but needs R_j and P_j has R_j but needs R_i.
  - Deadlock avoidance guarantees that every process makes progress and there is no process stuck waiting on a resource held by another process that is also waiting on a resource it can't get.
- **Starvation** is when a process never gets a chance to run. Waiting is unbounded.
  - Usually, starvation is the result of policy such as priority scheduling.
  - As long as every process gets a chance to run, starvation is avoided.

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode. ← Hard one!!
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode



Algorithm for Process P_i

```
do {

    while (turn == j);  // In Pj while (turn == i);

            critical section

    turn = j;

            remainder section

} while (true);
```

Reasons for failure:

Ensures Mutual Exclusion and bounded waiting (P0 and P1 alternate access).

If P0 takes an unduly long time in its remainder section, P1 never gets into the critical section. Progress is not made for P1 because P0 never sets turn = I;

Algorithm for Process P_i

```
do {
     flag[j] = TRUE;     // In Pj, flag[i] = TRUE;
     while (flag[i]);    // while (flag[j]);

             critical section

     flag[j] = FALSE;

             remainder section

 } while (true);
```

Reasons for failure:

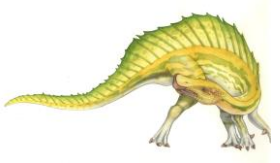
Satisfies Mutual Exclusion but not bounded waiting or progress.

Pi sets flag[j] = TRUE; // preempted

Pj sets flag[j] = TRUE; and enters loop while(flag[j]);

Pi enters while(flag[i]);

Neither can progress.



Algorithm for Process P_i

```
do {
     while (flag[i]);
     flag[j] = TRUE;
            critical section
     flag[j] = FALSE;
            remainder section
 } while (true);
```

Reasons for failure:

Satisfies progress but not bounded waiting. Mutual Exclusion problem

- Pj sets flag[i] = FALSE; remainder section gets preempted
- Pi leaves loop gets preempted
- Pj sets flag[i] = TRUE; in critical section gets preempted
- Pi sets flag[j] = TRUE; in critical section at the same time.

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!
- **NEED BOTH THE turn and flag[2] to guarantee Mutual Exclusion, Bounded waiting, Progress.**

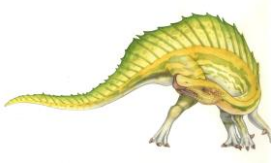
Peterson's Algorithm

Process P_i

```
do {
    flag[i] = true; // intent
    turn = j; // giving away
    while (flag[j] && turn==j)
        ; // wait for either
        critical section
    flag[i] = false;
    remainder section
} while (true);
```

Process P_j

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn==i)
        ;
        critical section
    flag[j] = false;
    remainder section
} while (true);
```



Algorithm for Process P_i

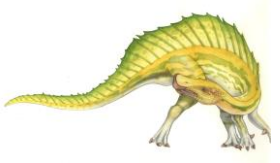
```
do {
    flag[i] = true; // i is ready
    turn = j; // Give Pj a chance first
    while (flag[j] && turn == j);
            critical section
    flag[i] = false;
            remainder section
} while (true);
```

Peterson's says, Pi is ready to enter critical section. Let Pj go first.

If Pj is waiting, then flag[i] == true but turn != i because turn was just set to j.. therefore, Pj proceeds out of loop to critical section and Pi is in the loop.

if Pj is not waiting, then flag[j] == false, so Pi can proceed into the critical section.

if Pj is in the critical section, then Pi waits because flag[j] == true (Pj set it) and turn == j (Pi set it). When Pj is done, flag[j] == false, so Pi can proceed.



Peterson's Solution

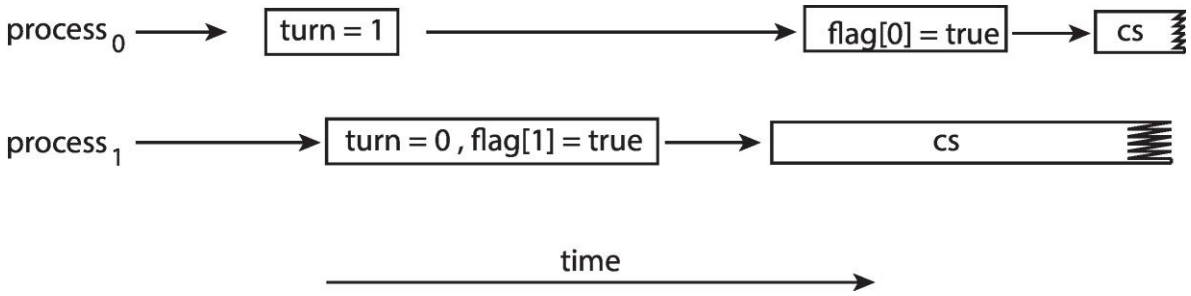
- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- Understanding why it will not work is also useful for better understanding race conditions.
- To improve performance, processors and/or compilers may reorder operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Peterson's Solution

- 100 is the expected output. Thread 1 has print x;
- However, the operations for Thread 2 may be reordered:

```
flag[1] = true;
x = 100;
```

- If this occurs, the output may be 0!
- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!

  This can happen merely from a race condition because setting turn and flag must be atomic or it doesn't work.

Bakery Algorithm (1)

- Critical Section for n processes:
  - Before entering its critical section, a process receives a number (like in a bakery). Holder of the smallest number enters the critical section.
  - The numbering scheme here always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first (PID assumed unique).

Bakery Algorithm (2)

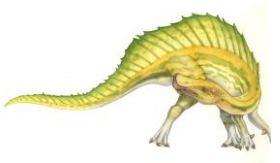
- Choosing a number (take a ticket):
  - Number[i] = 1 + max (a_0,…, a_{n-1}) is a number k, such that $k \geq a_i$ for i = 0, …, n – 1 where (a_0,…, a_{n-1}) is held by another thread.
- Notation for lexicographical order (ticket #, PID #)
  - (a,b) < (c,d) if $a < c$ or if $a == c$ and $b < d$
- Shared data:

  boolean choosing[n];

  int number[n];

  Data structures are initialized to **FALSE** and **0**, respectively.

To enter critical section: choose a ticket, use the PID to resolve ties. resource goes to next waiting process with lowest ticket number or (lowest ticket number, lowest PID).



Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

  1. Memory barriers

  2. Hardware instructions

  3. Atomic variables

Memory Barriers

- **Memory models** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.
  - A fence that states all memory accesses before the barrier be complete so that all accesses after the barrier have the order preserved.
  - Needed when accessing shared variables on current multicore systems

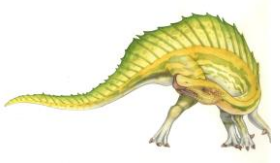
Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier(); // Make sure while before print x
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier(); //Make sure x = 100 before flag = TRUE
flag = true
```

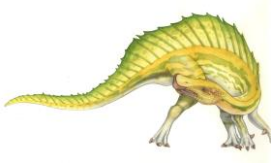


Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
  - Test-and-set / Swap(xchg on Intel) are **Atomic** operations.

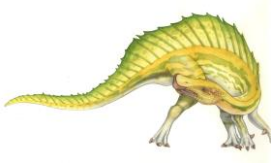


test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
   {
        boolean rv = *target;
        *target = TRUE;
        return rv:
   }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.
4. When target == FALSE, target is set to TRUE but FALSE is returned, so loop is exited.
5. When target == TRUE, target is set to TRUE, but TRUE is returned, so continue to loop.

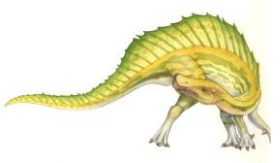


Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

    lock = false;

            /* remainder section */

 } while (true);
```

Think of test_and_set as an atomic version of i++ except it can only ever be 0 or 1. When leaving the critical section, lock is reset.



Test and Set Instruction

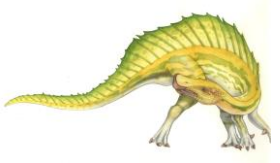
- Mutual exclusion is preserved:
- if Pi enters the CS, the other Pj are busy waiting
- • Problem: still using busy waiting
- • When Pi exits CS, the selection of the Pj who will enter CS is arbitrary: no bounded waiting. Hence starvation is possible
- • Processors (ex: Pentium) often provide an atomic xchg(a,b) instruction that swaps the content of a and b.
- • But xchg(a,b) suffers from the same drawbacks as test-and-set

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    // if *value is FALSE it is unlocked, set to locked
    if (*value == expected) // expected = 0; new_value = 1
       *value = new_value;
 return temp;  // return unlocked
}
```

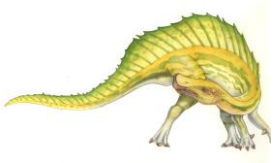
1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.



Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- When lock is 0, lock == expected so 0 is returned but lock is 1
  - Loops while lock == 1, when another process exits, sets lock = 0
  - Exits loop when lock == 0.
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
    /* critical section */
       lock = 0;
    /* remainder section */
} while (true);
```



Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;  // i wants to enter CS
   key = true;         // assume we are locked out
   while (waiting[i] && key) // busy wait for lock
      key = test_and_set(&lock);
   waiting[i] = false; // i no longer waiting, in CS

   /* critical section */

   j = (i + 1) % n;    // select next process j
   while ((j != i) && !waiting[j])
      j = (j + 1) % n; // does this j want to be next
   if (j == i)         // no other process waiting
      lock = false;    // unlock
   else                // otherwise, give j the lock
      waiting[j] = false; // let j in CS

   /* remainder section */
} while (true);
```

Entry

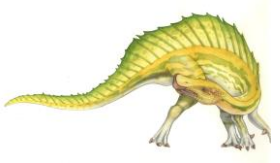
Exit

test_and_set Instruction

Definition:

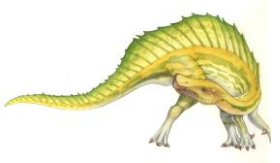
```
boolean test_and_set (boolean *target)
   {
        boolean rv = *target;
        *target = TRUE;
        return rv:
   }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.
4. When target == FALSE, target is set to TRUE to lock out others but FALSE is returned, so loop is exited.
5. When target == TRUE, target is set to TRUE, but TRUE is returned, so loop continues.



Bounded-waiting Mutual Exclusion with compare-and-swap

```
while (true) {
   waiting[i] = true;  // ENTRY SECTION
   key = 1;
   while (waiting[i] && key == 1)  // if lock == 0
      key = compare_and_swap(&lock,0,1); // set lock 1
   waiting[i] = false;             // but 0 returned

   /* critical section */

   j = (i + 1) % n;    // EXIT SECTION
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

   /* remainder section */
}
```



Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption:

```
increment(&sequence);
```

Atomic Variables

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
     int temp;

     do {
            temp = *v;
     }
     while (temp != (compare_and_swap(v,temp,temp+1));
}
```

```
int compare _and_swap(int *value, int expected, int new_value) {
        int temp = *value;
        // if *value is FALSE it is unlocked, set to locked
        if (*value == expected) // expected = 0; new_value = 1
           *value = new_value;
     return temp;  // return unlocked
    }
```

Machine Instruction Solution

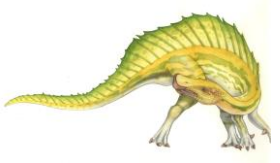
- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and easy to verify
  - It can be used to support multiple critical sections; each critical section can be defined by its own variable
- Disadvantages
  - Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting
  - Deadlock is possible if a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem – OS ensures it is atomic.
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
release() {
    available = true;
}
do {
    acquire lock
        critical section
    release lock
      remainder section
} while (true);
```



Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0) // 0 means first waiting, <0 means queue
        ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

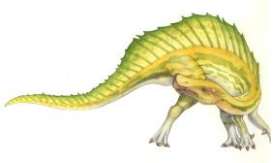
  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

If P1 gets there first, no problem.
If P2 gets there first, wait for synch.

Process P2 will busy wait for P1.

- Can implement a counting semaphore ***S*** as a binary semaphore



Semaphore Implementation

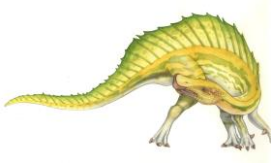
- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) { // Process has to wait
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) { // Processes are waiting
        remove a process P from S->list;
        wakeup(P);
    }
}
```



Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process, low priority never runs
  - Solved via **priority-inheritance protocol**

Priority Inversion Mars Pathfinder

- Tasks scheduled by VxWorks RTOS
  - Pre-emptive priority scheduling of threads.
- Threads scheduled according to urgency
- Meteorological data gathering task ran as an infrequent, low-priority thread and synchronized the information bus with a mutex.
- Higher priority tasks took precedence including
  - a medium priority, ***long running*** Communications Task,
  - a **very high** priority Bus Management Task.
- Low priority Meterological Task gets preempted by Communications Task in critical section.
- Bus Management Task preempts Communication Task but cannot make progress because mutex is held by Meteorological data gathering task. Blocks on mutex. Communication task continues.
- A watchdog timer wakes up, notices no Bus Management done, reboots all.

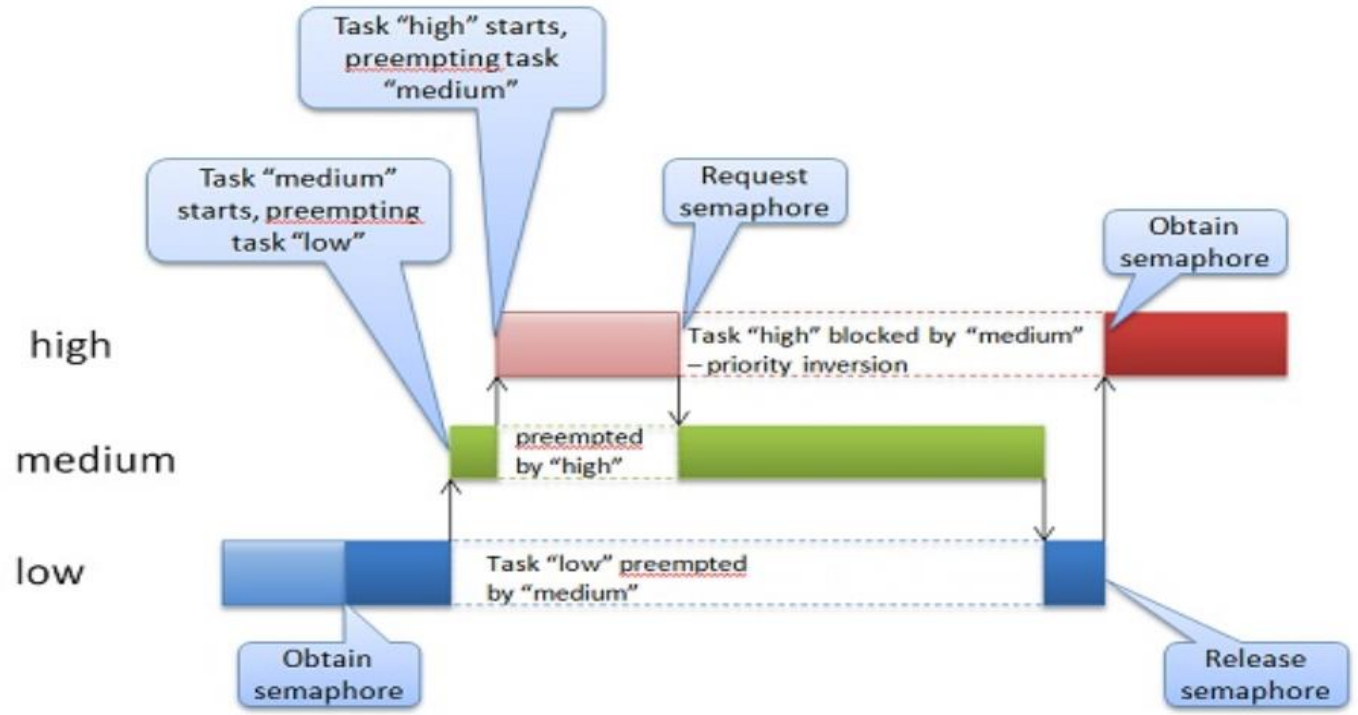
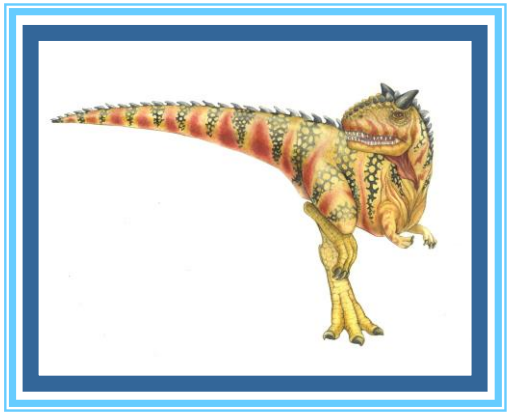


Figure 1: Priority inversion

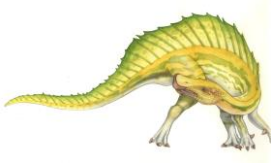
To fix the problem, they turned on a boolean parameter that indicates whether *priority inheritance* should be performed by the mutex. The mutex in question had been initialized with the parameter off; had it been on, the priority inversion would have been prevented.

Chapter 7: Synchronization Problems



Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem



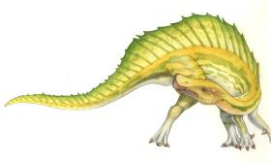
Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

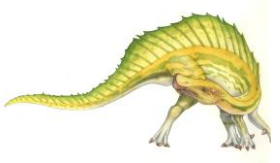
```
do {
        ...
        /* produce an item in next_produced, see slide 6 */
    wait(empty);
    wait(mutex);
        ... Assign the line read into the shared buffer
        /* add next produced to the buffer */
        ... And increment the shared count.4
            buffer[in] = inputString;
            in = (in + 1) % size;
            count ++;
    signal(mutex);
    signal(full);
    …. Fix the in index variable.
} while (true);
```



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
       ... Grab the next line out of the shared buffer
    /* remove an item from buffer to next_consumed */
    /* see slide 7 */
       ... Decrement the shared counter
           string ouput = buffer[out];
           out = (out + 1) % size;
           count --;
    signal(mutex);
    signal(empty);
       ... Cout the line and fix the out index
    /* consume the item in next consumed */
} while (true);
```



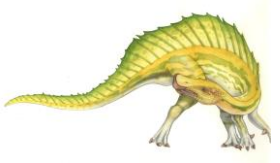
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 controls access to write.
  - Semaphore **`mutex`** initialized to 1 controls access to **read_count**
  - Integer **`read_count`** initialized to 0, keeps track of readers.

Readers-Writers Problem (Cont.)

- The structure of a writer process

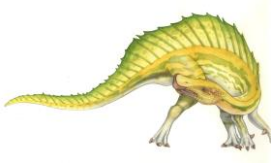
```
do {
        wait(rw_mutex);
          ...
        /* writing is performed */
          ...
        signal(rw_mutex);
} while (true);
```



Readers-Writers Problem (Cont.)

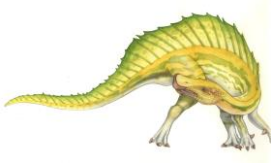
- The structure of a reader process

```
do {
    wait(mutex);          // Only 1 can access read_count
       read_count++;
       if (read_count == 1)
          wait(rw_mutex);// First reader blocks writer.
    signal(mutex);
          ...
       /* reading is performed */
          ...
    wait(mutex);
       read count--;
       if (read_count == 0)
          signal(rw_mutex);// No reader, unblock writer
    signal(mutex);
} while (true);
```



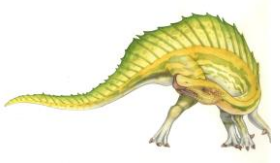
Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks



2nd Reader-Writers, Writers Priority

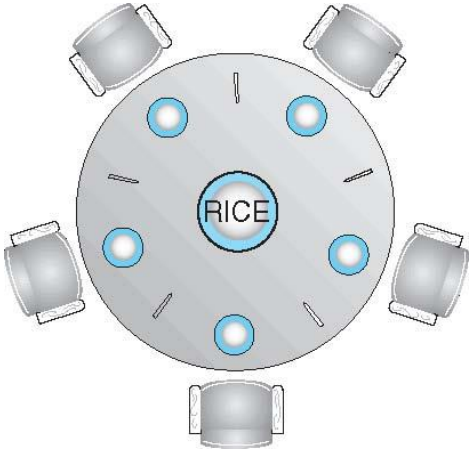
```
do {  // Reader process, extra mutex
     wait(read_mutex);
     wait(mutex);          // Only 1 can access read_count
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);// First reader blocks writer.
     signal(mutex);
     //signal(read_mutex); Is this necessary?
         ...
        /* reading is performed */
         ...
     wait(mutex);
        read count--;
        if (read_count == 0)
          signal(rw_mutex);// No reader, unblock writer
     signal(mutex);
 } while (true);
```



Writer Process, writer priority

```
do {  // Writer process similar to reader, also uses extra mutex
        wait(mutex);                // Only 1 can access write_count
           write_count++;
           if (write_count == 1)
               wait(read_mutex);// First writer blocks readers.
        signal(mutex);
        wait(rw_mutex);             // A writer has access
            ...
           /* writing is performed */
            ...
        signal(rw_mutex);
        wait(mutex);
           write_count--;
           if (write_count == 0)
             signal(read_mutex);// No writer, unblock writer
        signal(mutex);
    } while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

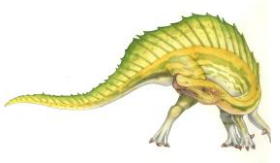
- The structure of Philosopher *i*:

```
do {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

                    //  eat

        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

                    //  think

} while (TRUE);
```

- What is the problem with this algorithm?

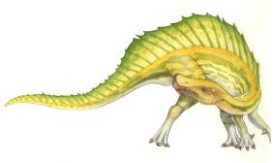
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table. Leaves an extra.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.



Problems with Semaphores

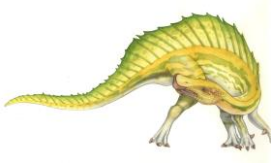
- Semaphores are complicated and hard to get right.
  - Extremely error prone!
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.



Monitors

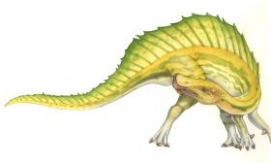
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization using mutual exclusion.
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time.
  - Great for ensuring mutual exclusion.
  - Monitor keeps track of waiting threads/processes.
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

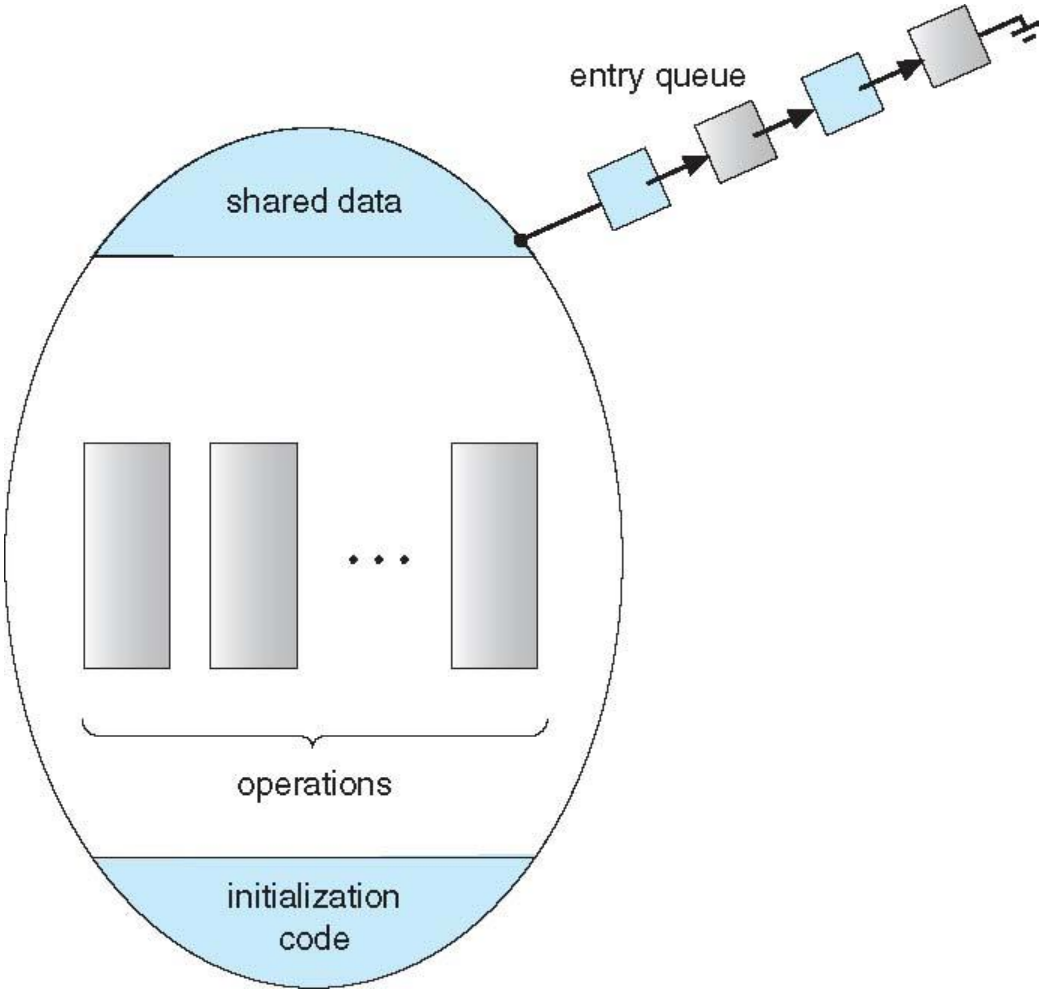


Monitors – Abstract Data Type

- Has an initialization section which runs once.
- Shared data that is often the reason for synchronization.
- One or more operations that access the shared data.
- A queue of waiting tasks (threads/processes).
- A simple monitor cannot control inter-task dependencies such as
  - T1 must perform some operation which is required before T2 can proceed.

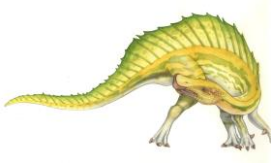


Schematic view of a Monitor

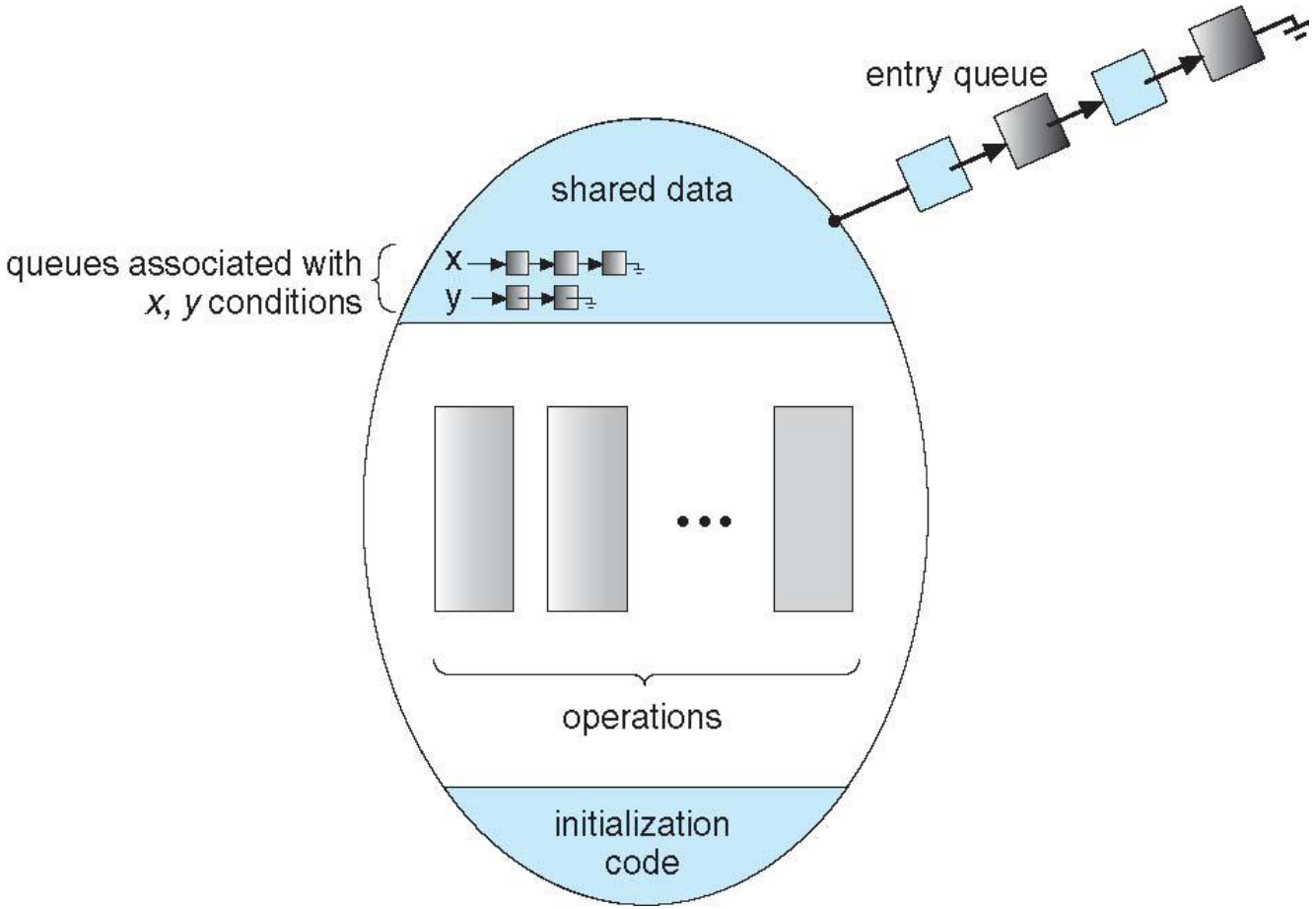


Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable (different from semaphores)
- If a thread issues **x.wait()** on condition **x**, if another thread has the resource, then the thread blocks and is put on the queue for x.
  - Another thread now has access to the monitor.

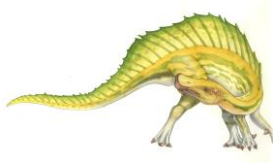


Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in parallel. One must continue, but which one?
- Options include
  - **Signal and wait** – P waits, Q continues until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – P continues until it either leaves the monitor or it waits for another condition, Q waits for P to relinquish monitor
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

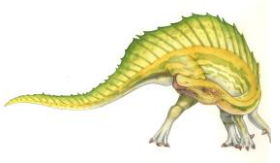


Monitor Solution to Dining Philosophers

- Problem: Philosophers all sitting around a table with one chopstick between each pair.
- Solution is for a Philosopher to only pick up chopsticks when the two closest chopsticks are available.
- Means that the Philosophers on either side do not have the chopsticks.
  - Monitor is helpful because both conditions can be checked and synchronized
- A Philosopher can either be: **HUNGRY**, **THINKING** or **EATING**.
- If a Philosopher is **HUNGRY**, request access to the chopsticks on either side:
  - Must test if Philosophers on either side are **NOT EATING**.
  - If a Philosopher is running a function of the Monitor, that Philosopher has exclusive access to the Monitor.

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING, HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i); // Tests if chopsticks are available
          if (state[i] != EATING) self[i].wait;
    }

   void putdown (int i) {
          state[i] = THINKING;
                  // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
    }
```



Solution to Dining Philosophers (Cont.)

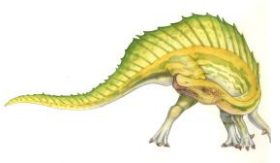
```
    void test (int i) {
            // both chopsticks must be available L/R
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ; // Gets chopsticks
             self[i].signal () ;
            }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
    }
}
```

Solution to Dining Philosophers (Cont.)

- Initialization
- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

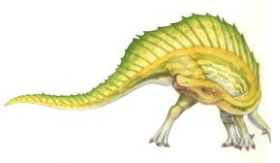


Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do { // Can't just put a mutex around these.
     // need to test both atomically for availability
     // need to be able to grab them both, if not, wait
     wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                  //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                    //  think
} while (TRUE);
```

- What is the problem with this algorithm?



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
  body of F;
    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem;     // (initially  = 0)
int x_count = 0;
```

- Implementation Q waits for condition x, signals waiting P or leave open.
- The operation x.wait can be implemented as:

```
x_count++;           // May end up waiting
if (next_count > 0)  // Something waiting?
   signal(next);     // signal next P to go
else                 // nothing waiting.
   signal(mutex);    // open monitor
wait(x_sem);         // wait for x_sem
x_count--;           // No longer waiting
```

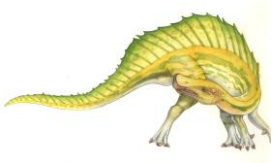
Monitor Implementation (Cont.)

- Implementation P waits when Q continues because condition x is met.
- The operation **`x.signal`** can be implemented as:

```
if (x_count > 0) { // something waiting
   next_count++;    // Current may end up waiting
   signal(x_sem);   // Schedule next P for x_sem
   wait(next);      // Current now waiting
   next_count--;    // P released, no longer wait
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next
  - Remember to avoid priority inversion with priority inheritance protocol. What's this?



Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
    ...
  access the resurce;
    ...

R.release;
```

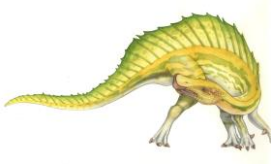
- Where R is an instance of type **`ResourceAllocator`**
- Allocate the resource to the process that requests the shortest time.
- Back to the problem where programmers have to manage **acquire** and **release** calls which is error prone.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
   }
   void release() {
            busy = FALSE;
            x.signal();
   }
initialization code() {
    busy = FALSE;
   }
}
```

Synchronization Examples

- Windows
- Linux
- Pthreads



Windows Synchronization

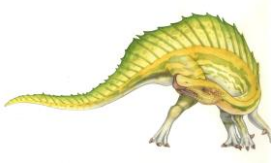
- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** in user-land which may act as mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both (atomic integers and spinlocks).
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

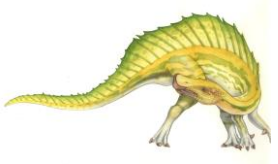
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks



Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages



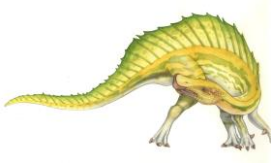
Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()
 {
        acquire();
        /* read/write memory */
        release();
 }
```

Mimics a transactional database. Keeps a log on changes that can be either committed or roll-back depending on if the transaction finishes. Not interrupted during atomic operation below.

```
void update ()
{
    atomic { /* transactional */
          /* read/write memory */
     }
}
```



Transactional Memory

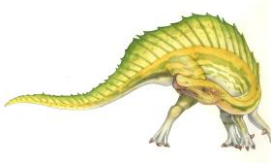
- Transactional memory system is responsible for guarding the memory access.
- Relieves the programmer from responsibility of complex synchronization code with locks and semaphores.
- Can be software or hardware based.
- **Software transactional memory** compiler inserts instrumentation code inside transaction blocks and can determine where concurrent access can be implemented and where low-level locking is required.
  - No hardware support needed.
- **Hardware transactional memory** uses hardware cache hierarchies and cache coherency protocols to manage conflicts and involving shared data
  - No instrumentation needed, lower overhead than STM.
- STM/HTM has been around a while, multicore processors have motivated research into multiprocessor applications.

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
        #pragma omp critical
        {
              count += value
        }
  }
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.



Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- Eliminates side-effects. A function f(a) called with arg a will always have the same result regardless of when it is called.
- Lazy evaluation - synchronous mechanism in a concurrent environment. Lightweight, easy way to create a reference to a computation and share the results among many threads.
- If multiple threads attempt to access an unevaluated expression, one will execute it while the others block. In this way, synchronization is built-in.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.
  - Haskell
  - Clojure

Bounded Buffer Lab

- Initialize three semaphores mutex, full and empty. Using the sem_init call, set the **mutex** to 1, **full** to 0 and **empty** to buffer size.
- The simplest implementation would have the semaphores as global variables that can be accessed by any function. You can make the **in** and **out** indices global, as well as the **string array**.
- You also need a ifstream for reading the file. In both producer and consumer, test for infile is not EOF.

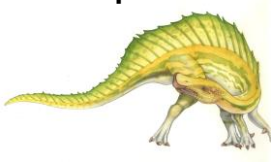
  while (getline(infile, next)) // this will stop on eof.
- The producer will be ahead of the consumer, most likely. Leave a space free in the buffer. It makes it easier to stop.

  producer: in = (in + 1) % (size – 1); // set buffer[in], increment count

  consumer: out = (out + 1) % (size – 1); // take buffer[out], decrement

  Use a counter++ in producer and counter– in consumer.

  If there are no remaining strings in the buffer, you want to exit the loop.



End of Chapter 7